Introduction

- Thesis
  - Vista protections are largely ineffective at preventing browser exploitation

- Overview
  - Whirlwind tour of Vista protection mechanisms
    - GS, SafeSEH, DEP, ASLR
  - Techniques for exploiting protection limitations
    - All protections broken
  - Conclusion

- Full paper available at http://www.phreedom.org/research/
Demo

- Exploiting IE despite all protections on Vista
  - ASLR and DEP turned on
  - Third party plugins NOT required for exploitation
- This works with IE8 as well

Part II:
Vista Protection Features

Memory Protection Mechanisms

<table>
<thead>
<tr>
<th></th>
<th>XP SP2, SP3</th>
<th>FP1, FP2</th>
<th>Vista SP1</th>
<th>Vista SP2</th>
<th>Vista SP1</th>
<th>Vista SP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>variable reordering</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>explorer_strong_check</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IIoM</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ASLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack cookies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>heap protection</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>page table modification</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>heap metadata cookies</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>heap metadata encryption</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DEP</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>NRE support</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>protected DEP</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Optimal mode by default</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ASLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peb, TEB</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>heap</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>stack</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>regress</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Memory Protection Mechanisms

• Detect memory corruption:
  – GS stack cookies
  – SEH chain validation
  – Heap corruption detection

• Stop common exploitation patterns:
  – GS (variable reordering)
  – SafeSEH
  – DEP
  – ASLR

GS Stack Cookies

• GS prevents the attacker from using an overwritten return address on the stack
  – Adds a stack cookie between the local variables and return address
  – Checks the cookie at the function epilogue

GS Variable Reordering

• Prevents the attacker from overwriting other local variables or arguments
  – String buffers go above other variables
  – Arguments copied below local variables

<table>
<thead>
<tr>
<th>Source code</th>
<th>Standard stack frame</th>
<th>Stack frame with /GS</th>
</tr>
</thead>
<tbody>
<tr>
<td>void vuln(char* arg)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char buf[100];</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int i;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcpy(buf, arg);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td>return address arg</td>
<td>copy of arg</td>
</tr>
<tr>
<td>buf</td>
<td>stack cookie return address arg</td>
<td>(unused)</td>
</tr>
</tbody>
</table>
SafeSEH

- Prevents the attacker from using an overwritten SEH record. Allows only the following cases:
  - Handler found in SafeSEH table of a DLL
  - Handler in a DLL linked without /SafeSEH
- If DEP is disabled, we have one more case:
  - Handler on a non-image page, but not on the stack

SEH Chain Validation

- New protection in Windows Server 2008, much more effective than SafeSEH
  - Puts a cookie at the end of the SEH chain
  - The exception dispatcher walks the chain and verifies that it ends with a cookie
  - If an SEH record is overwritten, the SEH chain will break and will not end with the cookie
- Present in Vista SP1, but not enabled

Data Execution Prevention (DEP)

- Prevents the attacker from jumping to data:
  - Uses the NX bit in modern CPUs
  - Modes of operation
    - OptIn – protects only apps compiled with /NXCOMPAT. Default mode on XP and Vista
    - OptOut – protects all apps unless they opt out. Default mode on Server 2003 and 2008
    - AlwaysOn/AlwaysOff – as you’d expect
  - DEP is always enabled for 64-bit processes
    - Internet Explorer on Vista x64 is still a 32-bit process with no DEP
Data Execution Prevention (DEP)
- Can be enabled and disabled at runtime with NtSetInformationProcess()
  - Skape and Skywing’s attack against DEP
  - Permanent DEP in Vista
- Important: DEP does not prevent the program from allocating RWX memory

Address Space Layout Randomization (ASLR)
- Dramatically lowers exploit reliability
  - Relies on nothing being statically placed
- Several major components
  - Image Randomization
  - Heap Randomization
  - Stack Randomization
  - PEB/TEB Randomization

Address Space Layout Randomization (ASLR)
- Binaries opted-in to ASLR will be randomized
  - Configurable: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages
- Strategy 1: DLL randomization
  - Random offset from 0x79000000 up to 16M chosen (“Image Bias”)
  - DLLs packed together near the top of memory (First DLL Ending with Image Bias)
  - Known DLLs order also mixed up at boot time
  - Constant across different processes (mostly..)
- Strategy 2: EXE randomization
  - Random image base chosen within 16M of preferred image base
  - DLLs also use this strategy if "DLL Range" is used up
- Granularity of Address Space: 64K
Address Space Layout Randomization (ASLR)

- **Heap randomization strategy**: Move the heap base
  - Address where heap begins is selected linearly with NtAllocateVirtualMemory()”
  - Random offset up to 2M into selected region is used for real heap base
  - 64K alignment

- **Stack randomization strategy**: Selecting a random “hole” in the address space
  - Random 5-bit value chosen (X)
  - Address space searched X times for space to allocate the stack

- **Stack base also randomized**
  - Stack begins at random offset from selected base (up to half a page)
  - DWORD aligned

---

Part III: Breaking Vista Protections
**GS: Function Heuristics**

- **Functions containing certain types of variables are not protected:**
  - structures (ANI vulnerability)
  - arrays of pointers or integers

```c
void func(int count, int data) {
    int array[10];
    int i;
    for (i = 0; i < count; i++)
        array[i] = data;
}
```

**GS: Use of Overwritten Data**

- The function might use overwritten stack data before the cookie is checked:

<table>
<thead>
<tr>
<th>callee saved registers</th>
<th>copy of pointer and string buffer arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables</td>
<td></td>
</tr>
<tr>
<td>string buffers</td>
<td>o</td>
</tr>
<tr>
<td>gs cookie</td>
<td>v</td>
</tr>
<tr>
<td>exception handler record</td>
<td>e</td>
</tr>
<tr>
<td>saved frame pointer</td>
<td>r</td>
</tr>
<tr>
<td>return address</td>
<td>f</td>
</tr>
<tr>
<td>arguments</td>
<td>i</td>
</tr>
<tr>
<td>stack frame of the caller</td>
<td>w</td>
</tr>
</tbody>
</table>

**GS: Exception Handling**

- Triggering an exception will give us control of the program execution before the GS cookie check.
  - overwrite a pointer or counter variable
  - overflow to the top of the stack
  - application specific exceptions

- SEH records on the stack are not protected by GS, but we have to bypass SafeSEH.
Bypassing SafeSEH

- If DEP is disabled, we can just point an overwritten SEH handler to the heap

- If DEP is enabled, SafeSEH protections can be bypassed if a single unsafe DLL is loaded
  - Flash9f.ocx

DEP OptIn

- Vista runs in opt-in mode by default
  - Applications need to specifically opt-in to receive DEP protections

- No need to bypass something that isn’t there..
  - DEP not enabled in IE7 or Firefox 2
  - IE8 and Firefox 3 opted-in

ASLR OptIn

- Vista randomizes only binaries that opt-in
  - A single non-randomized binary is sufficient to bypass ASLR (and DEP)

- Some major 3rd party plugins do not opt-in
  - Flash
  - Java

- Microsoft does not utilize ASLR for all binaries
  - .NET runtime!
Heap Spraying

- **Heap spraying**
  - JavaScript (bypasses ASLR)
  - Java (bypasses ASLR and DEP)

- **Heap spraying can bypass ASLR**
  - Consume large amounts of address space with controllable data

- **Only the beginning of the heap is randomized**
  - The maximum offset is 2MB
  - If we allocate a chunk larger than 2MB, some part of it will be at a predictable address

- **JavaScript heap spraying**
  - Defeats ASLR (but not DEP)
  - 64KB-aligned allocations allow us to put arbitrary data at an arbitrary address
  - Allocate multiple 1MB strings, repeat a 64KB pattern
Heap Spraying - Java

- The Sun JVM allocates all memory RWX
  - DEP not an issue
  - ASLR mitigated

Executable heap spraying code:

```java
public class Test extends Applet {
  static String foo = new String("AAAA...");
  static String[] a = new String[50000];
  public void init() {
    for (int i = 0; i < 50000; i++) {
      a[i] = foo + foo;
    }
  }
}
```

Heap Spraying - Java

- Screenshot

```
0:031> !vadump
BaseAddress: 22cc0000
RegionSize: 058a0000
State: 00001000 MEM_COMMIT
Protect: 00000040 PAGE_EXECUTE_READWRITE
Type: 00020000 MEM_PRIVATE
```

Stack Spraying

- Alternative to heap spraying
  - High degree of control over stack contents
  - Creating pointers is simple too: objects/arrays/etc as parameters/local variables
  - Also usable to exhaust large parts of the address space

- Stack size is controlled by the attacker in .NET and Java!
  - Thread constructors allow stack size of your choosing
Stack Spraying

**Method 1: Generate Code**
- Large amount of local variables
- Fill with executable code
- DEP will prevent execution, but this is also true of heap spraying

**Method 2: Overwrite Targets**
- Fill the stack with useful pointers to overwrite
- Saved EIPs are probably most useful
- Create a recursive function to fill the entire stack
- Overwrite anywhere in the memory region for the win!
Stack Spraying

- **Method 3: Pointer Spraying**
  - Languages don’t allow pointer creation directly
  - Declaring objects/arrays will create pointers
  - Useful for exploits requiring indirection

.NET and IE

- **IE allows embedding of .NET “User Controls”**
  - .NET equivalent of a Java applets
  - Embedded in a web page using the `<OBJECT>` tag
    `<OBJECT classid="ControlName.dll#Namespace.ClassName">`
  - Unlike ActiveX, no warning in “Internet Zone”

- **User controls are .NET DLLs**
  - That’s right — DLLs can be embedded in web pages!
  - Similar to native DLLs with some additional metadata
  - They can’t contain native code (IL-Only)
  - Loaded into the process with LoadLibrary
.NET shellcode

- Loading User Controls is interesting in the context of memory protections
  - We can define memory region sizes
  - Page protections are arbitrary
  - In XP, Image base is directly controllable by the attacker
  - On Vista, ASLR prevents direct load address control
    - IL-Only binaries are always randomized, despite opting out of ASLR
    - Load address can still be influenced

.NET Controls - Large DLLs

- Large DLL Method 1
  - Create a large DLL (~100MB)
  - Must consume less than “Standard DLL range”
  - Approximate load location easily guessable

- Large DLL Method 2
  - Create even larger DLL (~200MB)
  - Approximate load location easily guessable
  - Additional bonus: Select addresses that will bypass character restrictions
**.NET Controls - Large DLLs**

- **Problem:** 100M+ is too much to download
  - Pages will take too long to load
- **Solution 1: Binary Padding**
  - For a given section, make the VirtualSize very large, and SizeOfRawData 0 or small
  - Zero-padded when mapped
  - Repeating instruction “add byte ptr [eax], al”
  - Needs EAX to point to writable memory
- **Solution 2: Compression**
  - HTTP can zip up content on the fly
  - Achieved with Content-Encoding header

---

**.NET Controls - Large DLLs**

- **Large DLL Method 3**
  - Create large DLL (Virtual Padding)
  - Create smaller 16M DLL with shellcode etc
  - Compress smaller DLL with HTTP

---

**.NET Controls - Small DLLs**

- **Small DLL Method**
  - Embed a large number of small DLLs (4-8K)
  - About 300 of them is enough (~20M)
  - They all get placed on 64K boundaries in “Standard DLL Range”
  - Target any one of the DLLs in range
.NET Controls – Statically Located DLLs

• Ideal situation is to have statically positioned, self-supplied .NET DLLs
• ASLR enforced on IL-Only binaries
  – Loader checks if binary is an .NET IL-Only binary and relocates it anyway (no opting out)
  – Is this effective? Not quite…
• Flagging an IL-Only binary depends on version information read from .NET COR header!

Code from MiCreateImageFileMap():

```c
if( (pCORHeader->MajorRuntimeVersion > 2 ||
    (pCORHeader->MajorRuntimeVersion == 2 && pCORHeader->MinorRuntimeVersion >= 5)) &&
    (pCORHeader->Flags & COMIMAGE_FLAGS_ILONLY) )
{   
    pImageControlArea->pImageInfo->ImageBaseInfo->bFlags |= PINFO_IL_ONLY_IMAGE;
    ...
}
```

• Statically position DLL in 3 Simple steps
  – Opt out of ASLR (unset IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE)
  – Select ImageBase of your choosing
  – Change version in COR header (2.5 -> 2.4 is sufficient)

.Demo
Conclusion

Part IV: Conclusion

- Vista memory protections are ineffective at preventing browser exploitation
  - Large degree of control attacker has to manipulate process environment
  - Open plugin architecture
  - Single point of failure

- More work needed on secure browser architecture
  - Google Chrome is an interesting new development

- Questions?